Introduction à la programmation orientée objet 1

Version 2021/2022

Table des matières

1. Notions importantes....................7
1.1. Qu'est-ce qu'un « objet » ?....................7
1.2. Exemples d'introduction....................8
1.2.1. Exemple 1 : Voitures....................8
1.2.2. Exemple 2 : Personnes....................9
1.3. Qu'est-ce qu'une « classe » ?....................10
1.4. Qu'est-ce qu'un « attribut » ?....................12
1.5. Qu'est-ce qu'une « méthode » ?....................13
1.5.1. L'instruction *return*....................14
1.5.2. Le mot clé *void*....................14
1.6. Qu'est-ce qu'un « paramètre » ?....................17
1.7. Qu'est-ce qu'un « type » ?....................20
1.8. Conventions de noms pour les méthodes....................22
1.8.1. Les accesseurs - préfixe *get*....................22
1.8.2. Les manipulateurs - préfixe *set*....................22
1.8.3. Les méthodes effectuant un calcul - préfixe *calculate*....................22
1.8.4. Les méthodes booléennes - préfixes *is/has*....................22
1.8.5. La méthode *toString*....................22
1.9. Qu'est-ce qu'un « constructeur » ?....................23
1.10. Qu'est-ce qu'une « variable » ?....................24
1.11. Opérateurs, compatibilité et conversions....................26
1.11.1. Opérateur d'affectation « = »....................26
1.11.2. Opérateurs arithmétiques....................26
1.11.3. Conversions forcées de types (explicites)....................27
1.11.4. Conversions automatiques de types (implicites)....................27
1.11.5. Opérateur de concaténation « + »....................29
1.12. Affichage d'une ligne de texte....................29
1.13. La classe « Math »....................30
1.14. Générer des nombres aléatoires....................31
2. Présentation et documentation du code....................33
2.1. Qu'est-ce qu'un « commentaire » ?....................33
2.1.1. Règles pour l'utilisation des commentaires «JavaDoc»....................33
2.1.2. Précisions : les commentaires « JavaDoc » des classes....................34
2.1.3. Précisions : les commentaires « JavaDoc » des méthodes....................34
2.2. Qu'est-ce que « l'indentation » ?....................35
3. Les structures de contrôle....................36
3.1. La structure alternative....................36
3.2. La structure alternative et le retour de résultats....................38
3.3. Les opérateurs de comparaison....................39
3.4. Les opérateurs logiques....................40
3.5. La structure répétitive « tant que »....................41

3.6. La structure répétitive « pour »..........43
3.7. Les blocs et la durée de vie des variables..........45
4. Automatisation des tests des classes..........46
4.1. Création d'objets avec «new»..........46
4.2. *Automatiser les tests de fonctionnement*..........46
4.3. Démarrage automatique du programme..........47
4.4. Saisie de données au clavier en mode texte..........48
4.5. Pour avancés : Unimozer.monitor(...,...)..........49

Sources

- ***Introduction à Java, Robert Fisch, 2009***
- *Introduction à la programmation,* Robert Fisch, 11TG/T0IF, 2006
- *Introduction à la programmation,* Simone Beissel, T0IF, 2003
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006
- *Programmation en ANSI-C*, Fred Faber, T3IF, 1993-2006

Adaptations et actualisations

Fred Faber

Contributeurs

Gilles Everling, Guy Rhein, Claude Sibenaler

Site de référence

http://java.cnpi.lu

A la base de ce cours est le document « *Introduction à Java* » de Robert Fisch.

Le cours est adapté et complété en permanence d'après les contributions des enseignants.

Les enseignants sont priés d'envoyer leurs remarques et propositions via Mail à Fred Faber ou Robert Fisch qui s'efforceront de les intégrer dans le cours dès que possible.

La version la plus actuelle est publiée sur le site : **http://java.cnpi.lu**

Des ressources supplémentaires pour enseignants sont publiés sur l'espace de travail Informatique de *eduDocs*.

Introduction

Ce cours d'introduction à la programmation suit le principe "**objects first**" - "les objets d'abord" et ceci dans un double sens :

- au centre des intérêts se trouve la programmation orientée objet : tous les exemples ou exercices sont traités du point de vue orienté objet,
- le cours commence par l'introduction des principes et du vocabulaire orienté objet avant même de traiter les opérateurs et les structures de programmation.

Dans le cours de 3e, nous allons développer des classes et manipuler des objets, sans pour autant nous occuper de l'interface ni de la saisie ou de la représentation des données.

Le logiciel Unimozer joue le rôle de "banc d'essai" pour nos objets et nous permet de vérifier le bon fonctionnement de nos classes sans que nous n'ayons besoin de développer des applications autour de nos classes. Ainsi Unimozer crée automatiquement des dialogues pour demander les valeurs et pour afficher des résultats.

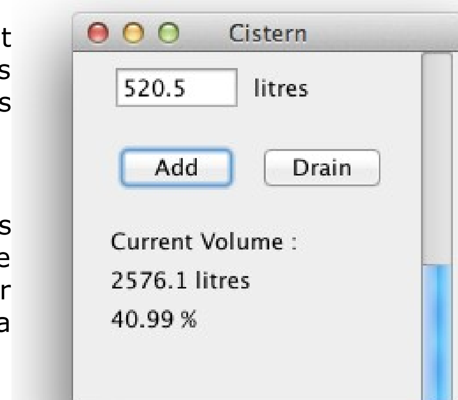
Exemple : Le projet 'Cistern' en 3e

Cistern
− maximumVolume : double
− currentVolume : double
+ Cistern(pRadius : double, pHeight : double)
+ add(pVolume : double) : void
+ drain(pVolume : double) : void
+ getCurrentVolume() : double
+ getCurrentRate() : double
+ toString() : String

void add(double)
pVolume (double) = 520.5
Cancel OK

cistern0 : Cistern
maximumVolume = 6283.185307179586
currentVolume = 0.0

Lors du test de la méthode **add**, c'est Unimozer qui crée automatiquement un dialogue pour nous demander le nombre de litres à ajouter à la citerne.



Dans le cours de 2e, nous allons voir comment développer des applications avec des interfaces graphiques individuels pour visualiser et manipuler nos objets.

Exemple : Le projet 'Cistern' en 2e

Ce programme utilise la classe *Cistern* que nous avons développée en 3e, mais en 2e nous saurons construire une interface plus conviviale pour présenter et changer le contenu de la citerne. Ainsi, les détails de la réalisation resteront cachés à l'utilisateur.

Conventions :

Dans ce cours, nous allons utiliser des désignations anglaises pour tous les éléments que nous définissons dans le code de nos classes (noms des classes, attributs, méthodes, variables, ...).

Ainsi, nous évitons :

- d'une part des aberrations linguistiques (p.ex: **`getValeur`**, **`setMoyenne`**),
- d'autre part des incompatibilités dues par exemple à des caractères accentués.

De préférence, les textes retournés par nos objets ou affichés à l'écran seront des textes anglais. Les commentaires dans notre code seront de préférence en français et serviront donc aussi à expliciter les désignations anglaises moins connues.

1. Notions importantes

OOP *- **O**bject **O**riented **P**rogramming - **o**bjekt**o**rientierte **P**rogrammierung*

POO *- **P**rogrammation **o**rientée **o**bjet*

Dans la programmation Java, nous allons manipuler des « objets » dont les « attributs » et les « méthodes » sont définis dans des « classes ». Dans ce chapitre, nous allons essayer d'éclaircir ce vocabulaire qui constitue le fondement de la programmation orientée objet.

1.1. Qu'est-ce qu'un « objet » ?

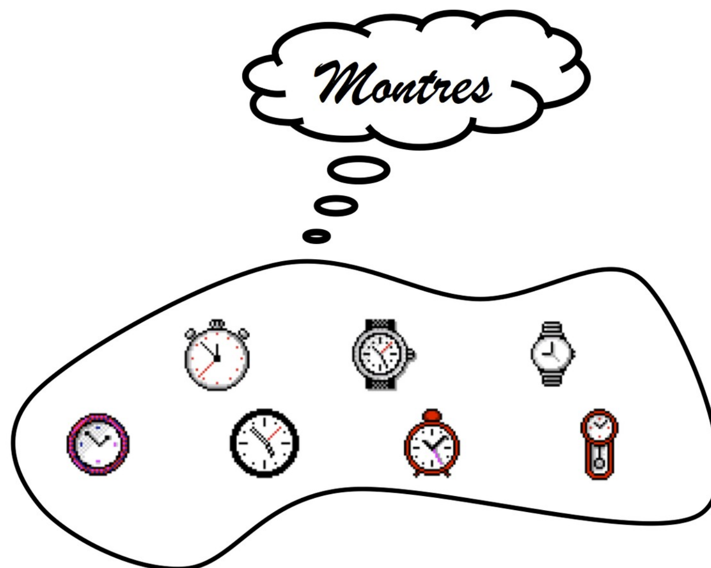
Dans la vie courante, toute chose, vivante ou non, peut être appelée *objet*. Les « objets » peuvent être une montre, un thermomètre, un client, un rendez-vous par exemple.

Nous pouvons définir un objet informatique comme suit :

> Un **objet** est la représentation informatique d'un individu, d'un objet ou d'un concept de la vie réelle. Il est défini par ses caractéristiques propres (appelés « *champs* » ou « ***attributs*** *»)* et par son comportement (ses « ***méthodes*** *»)*.

Cependant, chaque individu décrit un même objet différemment parce que chaque observateur a une autre personnalité, d'autres préférences, d'autres priorités, un autre point de vue.

Ainsi, chacun attribue d'autres importances aux caractéristiques d'un objet. On dit que chaque personne a une perception différente de la même chose. L'homme crée donc une image abstraite d'un objet dans sa tête. On dit qu'on a fait **abstraction**.



> **L'abstraction** est le fait d'ignorer les propriétés marginales et de se concentrer sur les aspects essentiels. C'est un principe essentiel pour pouvoir gérer la complexité du monde réel dans le domaine informatique.

En informatique, l'abstraction nous permet de ne considérer que les caractéristiques les plus importantes des objets que nous manipulons.

1.2. Exemples d'introduction

Avant de passer aux définitions plus formelles du vocabulaire de la POO, discutons quelques exemples pour comprendre l'approche de la modélisation d'objets dans la programmation.

1.2.1. Exemple 1 : Voitures

Trois personnes, Monsieur X, Madame Y et Monsieur Z, possèdent chacun leur propre voiture. Chacune de leurs voitures a 4 roues, un moteur, une certaine vitesse de pointe et ainsi de suite. De même, chacune de ces voitures peut accélérer, freiner, changer de direction, etc. On voit donc que chacune de ces trois voitures a le même type **d'attributs** et de **comportements** ou fonctionnalités. C'est pour cela qu'on dit que ce sont des voitures.

Et pourtant, les trois voitures ne sont pas identiques. En fait, voici une comparaison des trois voitures :

Propriétaire	Monsieur X	Madame Y	Monsieur Z
Peinture	verte	blanche	noire
Poids	1.500 kg	1.200 kg	1.700 kg
Puissance	100 kW	75 kW	170 kW
Vitesse de pointe	190 km/h	180 km/h	254 km/h
Accélérer	oui	oui	oui
Freiner	oui	oui	oui
Changer de direction	oui	oui	oui

Nous pouvons dire que les 3 véhicules partagent les mêmes types de caractéristiques d'un point de vue conceptionnel :

- chaque véhicule a un propriétaire, une certaine peinture, un certain poids, une certaine puissance et une certaine vitesse de pointe (→ **attributs**)
- chaque véhicule peut accélérer, freiner et changer de direction (→ **méthodes**).

Il y a bien sûr de nombreuses autres caractéristiques que nous allons ignorer dans cet exemple, afin de limiter la complexité (→ **abstraction**).

Même si les valeurs des attributs sont individuelles, les trois objets sont quand même des objets similaires qui ont la même identité : ce sont des voitures (et non pas des téléphones, des chaises ou des pommes).

On peut donc dire qu'il s'agit de trois **objets** différents, qui appartiennent tous à la même **classe**, appelée « Voiture ». En d'autres mots:

La classe décrit le type d'objet.

Un objet concret et spécifique est appelé une instance (une réalisation) d'une classe.

Ainsi l'objet « voiture de Monsieur X » est une instance de la classe « Voiture ». Il en est de même pour les deux autres voitures.

1.2.2. Exemple 2 : Personnes

Dans votre salle de classe, il y a un nombre d'individus, d'« objets vivants ».

- Chacun de ces objets a 2 bras, 2 jambes, une tête, une personnalité, un certain âge, un lieu de résidence, etc.
- De même, chacun de ces objets peut marcher, s'asseoir, courir, rire, parler, etc.

Il y a toute une série de types **d'attributs** et de comportements ou de **méthodes** que ces **objets** ont en commun. On peut donc qualifier tous les objets vivants dans cette salle d'êtres humains. On dit que ces objets sont des **instances** de la **classe** « être humain ».

On remarque à nouveau que chacun de ces objets se distingue des autres. Ainsi nous avons tous les mêmes types d'attributs et de méthodes, mais nos attributs n'ont pas toujours la même valeur. P. ex. certains ont des cheveux bruns, d'autres des cheveux noirs, d'autres des cheveux blonds. Nous avons des tailles et personnalités différentes, etc. Nous sommes donc tous des objets différents, mais nous faisons partie de la même classe.

Reprenons maintenant le vocabulaire de façon plus formelle et intéressons-nous à la pratique de la POO…

1.3. Qu'est-ce qu'une « classe » ?

Une **classe** est une description formelle d'une collection d'objets similaires, de même identité.

Prenons l'exemple d'une classe dénommée **Ball**.

En **UML (Unified Modeling Language)**, la **classe** est représentée par un diagramme de classe :

Le code source y relatif est le suivant :

```
public class Ball
{
}
```

Explications :

- Le mot clé « **public** » indique qu'il s'agit d'une classe publique, c'est-à-dire d'une classe qui est utilisable par d'autres programmes ou d'autres classes.
- Le mot clé « **class** » indique qu'il s'agit d'une classe.
- Le mot «**Ball**» est le nom de cette classe. Par convention la première lettre d'un nom de classe en Java est écrite en majuscules et le reste en minuscules. Des caractères spéciaux ne sont pas admis !
- Les accolades « { » et « } » indiquent le début, respectivement la fin de la classe.

Comme défini précédemment, une classe est une description formelle d'une collection d'objets de même identité. Il s'agit donc purement d'une description générale. **Un objet est une instance d'une classe, c'est-à-dire que l'objet est créé dans la mémoire de l'ordinateur en suivant exactement les définitions contenues dans la classe.** Tous les objets instanciés à partir d'une même classe possèdent le même fonctionnement.

Autre analogie pour clarifier ces deux vocables :

classe	correspond	au **plan** d'une maison
objet	correspond	à la **maison** construite selon le plan

En POO, nous allons donc définir d'abord une ou plusieurs classes. Lorsque nous voulons tester ou faire tourner un programme, nous devons créer une ou plusieurs instances de ces classes. Ce sont ces instances qui 'vivent' dans la mémoire de l'ordinateur et avec lesquelles nous pouvons interagir.

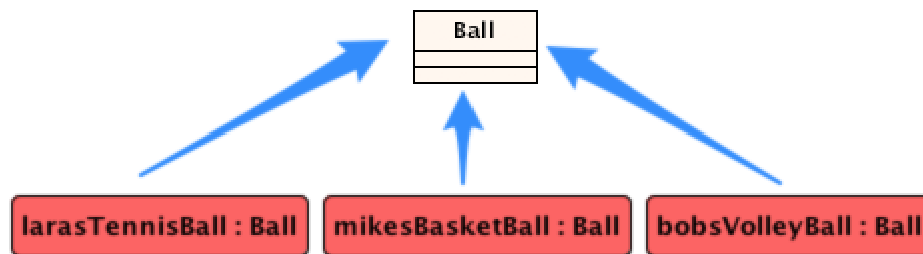
Comme désignations pour nos classes et objets, nous allons utiliser des noms ou des noms composés, p.ex. :

classes :	Person,	Client,	TennisBall,	SavingsAccount
objets :	person1,	client,	laurasTennisBall,	savingsAccount

En programmation, on dit qu'un **objet** est une « **instance »** d'une classe.

Le fait de créer un objet à partir d'une classe donnée s'appelle « **instanciation** ».

Reprenons l'exemple de la classe **Ball** :



Ball est la **classe** et contient la description abstraite (le plan) d'une balle dans le contexte de notre programme.

larasTennisBall, **mikesBasketBall** et **bobsVolleyBall** sont des **instances** de la classe **Ball**, donc des objets, c.-à-d. des balles concrètes avec leurs caractéristiques spécifiques (avec des valeurs différentes pour leur taille, leur couleur, leur pression).

L'environnement de développement (*Unimozer*)

Dans ce cours, nous utilisons un environnement de développement très simple (Unimozer) qui nous permet de construire des classes, de créer des instances, de voir leur contenu et de les tester.

Unimozer par exemple, affiche automatiquement la représentation UML correcte de la classe dans la partie gauche de l'écran. Le code Java se trouve dans la fenêtre à droite. L'outil actualise les données dans deux directions, c.-à-d. lorsque nous modifions le code Java, la représentation UML est actualisée automatiquement et vice-versa.

Exercice résolu :

- Si vous ne l'avez pas encore fait, alors créez un dossier pour vos exercices Java. Ouvrez votre interface de développement et ajoutez la classe publique **Ball**. Sauvez le projet sous '**00_Exercice_resolu**' dans votre dossier Java.
- Compilez la classe.
- Créez une nouvelle instance du nom de **larasTennisBall** de la classe **Ball**.
- Créez ensuite des nouvelles instances du nom de **mikesBasketBall** et **bobsVolleyBall**.

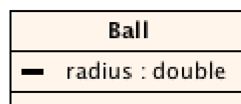
1.4. Qu'est-ce qu'un « attribut » ?

Un « **champ** » (angl. : ***field***) ou « **attribut** » d'une classe est une propriété qu'elle possède. Voici quelques exemples courants :

- La couleur d'une voiture est une propriété.
- La puissance dissipée d'une ampoule électrique est une propriété.
- La nationalité, âge et la taille sont des propriétés d'une personne.
- Le diamètre, le poids, la pression, la couleur sont les propriétés d'une balle.

Chaque attribut possède un certain **type** (→ voir chapitre 1.7) et un certain **nom**.

En UML, un attribut est inscrit dans la deuxième case d'un diagramme de classe. Voici un exemple de la classe **Ball** qui possède un attribut **radius** du type **double** (nombre réel) :



Le code source y relatif est le suivant :

```
public class Ball
{
    private double radius = 0;
}
```

Explications :

- Les attributs d'une classe doivent être protégés afin qu'uniquement les méthodes de la classe elle-même y aient accès. C'est pour cette raison que toutes les déclarations d'attributs que nous utilisons commencent par le mot clé « **private** ».
- L'attribut représente ici un nombre réel, donc du type **double** (→ voir chapitre 1.7).
- **radius** est le nom de l'attribut.
  Par convention les noms des attributs en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot (p.ex. **numberOfSides**).
- L'attribut **radius** est initialisé dès sa déclaration par la valeur zéro.
  Une déclaration d'un attribut sans initialisation se lirait comme suit :

```
private double radius;
```

Exercice résolu :

Ajoutez l'attribut **pressure** du type **double** à la classe **Ball.** Cet attribut contient la pression de la balle. Initialisez-le par la valeur 2,50. Compilez la classe et créez quelques objets. Que remarquez-vous lorsque vous inspectez les objets ?

- Étapes à suivre dans Unimozer :
  - clic droit sur l'objet
  - Add field …
  - puis entrez le nom et le type de l'attribut

1.5. Qu'est-ce qu'une « méthode » ?

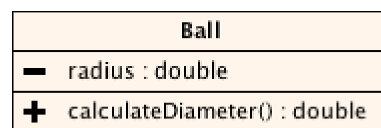
Une **méthode** est un sous-programme qui décrit une action que les instances de la classe savent effectuer. En termes informatiques, une méthode n'est rien d'autre qu'une suite d'instructions qui sont exécutées lorsqu'on fait appel à la méthode.

Une méthode peut retourner une valeur comme résultat ou ne rien retourner du tout (→ voir chapitre 1.5.2).

Le plus souvent, les méthodes changent le contenu des attributs de la classe ou utilisent les attributs de la classe pour effectuer une opération.

Exemple :

Donnons aux balles la capacité de nous fournir la valeur de leur diamètre :



En UML, une méthode est inscrite dans la troisième case d'un diagramme de classe :

En Java le code source y relatif est le suivant :

```
public class Ball
{
     private double radius = 11.95;

     public double calculateDiameter()
     {
          return 2 * radius;
     }
}
```

calculateDiameter est une méthode publique qui retourne une valeur décimale (du type double)

Explications :

- Le code source d'une méthode est indenté d'une tabulation par rapport au corps de la classe (→ voir chapitre 2.2).
- Le mot clé « **public** » indique qu'il s'agit d'une méthode publique, c'est-à-dire d'une méthode qui est utilisable par d'autres sous-programmes ou d'autres classes.
- Le mot clé « **double** » indique que la méthode retourne un nombre décimal (→ détails voir chapitre 1.7). En UML, le type du résultat est noté derrière le nom de la méthode (séparé par un « : »). Dans notre exemple: *calculateDiameter() : double*
- Le mot « **calculateDiameter** » est le nom de cette méthode.
  Les noms des méthodes en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot. Les noms des méthodes contiennent toujours un verbe.
- Les accolades « { » et « } » indiquent le début, respectivement la fin de la méthode.
- Une méthode peut avoir besoin de paramètres (→ voir chapitre 1.6) qui sont alors indiqués entre parenthèses derrière le nom de la méthode. Dans notre cas, la méthode ne possède pas de paramètres. Nous devons indiquer ce fait par une paire de parenthèses vide **()** derrière le nom.

1.5.1. L'instruction *return*

Une méthode qui ne travaille pas seulement avec les attributs, mais qui retourne en plus un résultat doit contenir l'instruction **return**. Le type du résultat à fournir doit être noté devant le nom de la méthode. L'instruction **return** se trouve en général à la fin de la méthode.

L'instruction **return** sert à :

1. **retourner comme résultat** de la méthode la valeur (ou le résultat de l'expression) qui se trouve derrière le mot clé **return**. Cette valeur ou expression doit nécessairement avoir le même type que la méthode.
2. **quitter** immédiatement la méthode.

1.5.2. Le mot clé *void*

Le mot clé **void** (DE: *leer* / FR: *vide*) est un cas spécial qui est uniquement utilisé lors de la déclaration de méthodes. Une méthode retourne **void** si elle effectue un travail sans pour autant retourner un résultat. Evidemment, une telle méthode ne possède pas d'instruction **return**.

Remarque :

Une méthode qui ne retourne pas **void** doit ***dans tous les cas*** se terminer par une instruction **return**. Si on oublie une instruction **return**, le compilateur s'arrête avec le message d'erreur *'missing return statement'*. Ceci jouera un rôle plus important lors du traitement des structures alternatives et répétitives (→ voir chapitre 3.1).

Exemple :

La classe suivante permet de calculer la circonférence de la balle et d'afficher ses informations:

```
public class Ball
{
  private double radius = 23.9;

  public double calculateCircumference()
  {
     return 3.1415926*2*radius;
  }

  public void printInfo()
  {
    System.out.println("I am a ball with a radius of " + radius + " cm");
    System.out.println("and a circumference of " + calculateCircumference() + " cm.");
  }
}
```

calculateCircumference est une méthode publique (sans paramètres) qui retourne une valeur décimale (type double)

printInfo est une méthode publique (sans paramètres) qui ne retourne pas de résultat, (type void), mais qui affiche un texte dans la fenêtre des messages

Observation importante :

Une méthode peut être appelée par d'autres méthodes. Ainsi on n'a pas besoin de programmer ou de recopier plusieurs fois les mêmes opérations. En fait, c'est l'utilité principale d'une méthode : le code devient plus compact, plus structuré et plus lisible.

Ici, la méthode **printInfo** fait appel à la méthode **calculateCircumference** pour obtenir la valeur de la circonférence de la balle. Le résultat retourné par la méthode **calculateCircumference** est intégré directement dans l'instruction d'affichage.

Discussion d'un exemple :

Créez la classe suivante:

```
public class Ball
{
    private double radius = 23.9;

    public double calculateCircumference()
    {
      return 3.1415926*2*radius;
      System.out.println("My circumference is " + 3.1415926*2*radius);
    }
}
```

Essayez de compiler la classe **`Ball`** et d'en créer une instance.

Notez vos observations et vos explications :

....

....

....

....

....

Pour avancés :

Quelle est la différence fondamentale entre l'instruction **`System.out.println`** et l'instruction **`return`** ? Que peut-on dire de l'utilité de l'instruction **`System.out.println`** à l'intérieur de la méthode **`calculateCircumference`**?

....

....

....

....

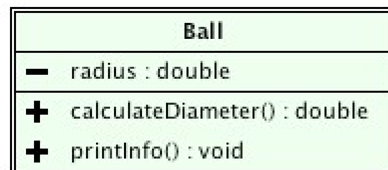
....

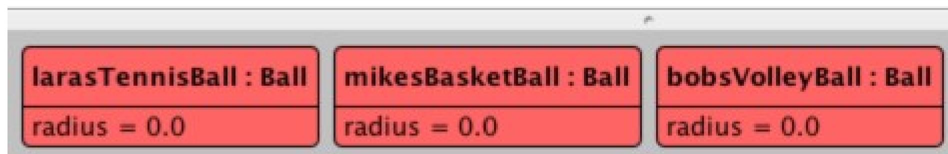
Exercice résolu :

- Ouvrez votre projet '**00_Exercice_resolu**' dans votre dossier Java.
- Ajoutez la méthode publique **`calculateDiameter`** comme définie ci-dessus.
- Ajoutez la méthode publique **`calculateCircumference`** sans l'instruction *System.out.println()*.
- Ajoutez la méthode publique **`printInfo`** comme définie ci-dessus.
- Compilez et créez une nouvelle instance du nom de **`mikesBasketBall`** de la classe **`Ball`**.
- Appelez les méthodes **`calculateDiameter`, `calculateCircumference`** et **`printInfo`** de l'objet **`mikesBasketBall`.**
- Répétez les deux points ci-dessus pour **`larasTennisBall`** et **`bobsVolleyBall`**. Que remarquez-vous?

Conclusions :

- Si la classe **`Ball`** possède un **attribut** **`radius`**, alors chaque instance de la classe (**`larasTennisBall`**, **`mikesBasketBall`**, **`bobsVolleyBall`**, …) possède cet attribut, mais la valeur de cet attribut peut être différente pour chaque instance :





(Nous allons voir dans les chapitres suivants comment donner des valeurs différentes aux attributs des différentes instances.)

- Si la classe **`Ball`** possède une **méthode** **`calculateCircumference`**, alors chaque instance de la classe (**`larasTennisBall`**, **`mikesBasketBall`**, **`bobsVolleyBall`**, …) possède cette même méthode. Ainsi la circonférence de toutes les balles est calculée d'après la même formule.

L'environnement de développement (*Unimozer)*

Lorsque nous appelons une méthode (qui ne retourne pas **`void`**) dans notre environnement de développement, alors le programme nous présente confortablement le résultat dans une fenêtre de dialogue. Voilà un luxe qui est spécifique à *Unimozer* et que nous ne trouvons pas dans d'autres environnements de développement. En général, le contenu des attributs se trouve caché dans la mémoire de l'ordinateur et les résultats fournis par une méthode passent de façon invisible d'une méthode à une autre. (Pour visualiser des valeurs, nous devrions employer l'instruction **`System.out.println`** ou programmer nous même une fenêtre de dialogue).

Ainsi nous pouvons considérer notre interface de développement comme une sorte de banc d'essai et d'analyse confortable qui nous permet de visualiser et de tester nos classes avant de les intégrer dans un programme fini.

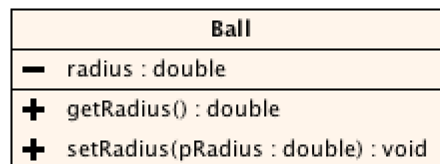
1.6. Qu'est-ce qu'un « paramètre » ?

Un **paramètre** est une donnée à l'aide de laquelle on peut passer une information à une méthode. Une méthode peut avoir aucun, un ou plusieurs **paramètres**.

Voici les caractéristiques d'un paramètre :

- il est d'un certain type (→ voir chapitre 1.7) et
- il possède un nom.

Donnons aux balles la possibilité de changer la valeur du radius :



Le schéma UML de la classe **Ball** peut être étendu de la manière suivante :

Voici le code source :

```
public class Ball
{
    private double radius = 0;

    public double getRadius()
    {
        return radius;
    }

    public void setRadius (double pRadius)
    {
        radius = pRadius;
    }
}
```

getRadius ne possède pas de paramètres

setRadius possède un paramètre du type double

void: la méthode ne retourne pas de résultat

Explications :

- Le mot clé «**public**» indique qu'il s'agit d'une méthode publique, c'est-à-dire d'une méthode qui est utilisable par d'autres sous-programmes ou d'autres classes.
- Le mot clé «**void**» indique que la méthode **setRadius** ne retourne pas de résultat.
- Le mot «**setRadius**» est le nom de cette méthode.
- «**double pRadius**» est la déclaration du **paramètre**. Dans notre cas il contiendra la nouvelle valeur pour le rayon:
    - « **double** » indique que le **type du paramètre** est un nombre décimal.
    - « **pRadius** » est le **nom du paramètre**. Comme les noms des méthodes et des attributs, les noms des paramètres en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot.
    - Pour mieux distinguer les paramètres des attributs, nous allons dans ce cours commencer les noms des paramètres par un 'p'.

Autre exemple :

Prenons la classe **Point** et ajoutons la possibilité de modifier sa position :

Point
− x : double − y : double
+ setCoordinates(pX : double, pY : double) : void

```
public class Point
{
     private double x = 0;
     private double y = 0;

     public void setCoordinates(double pX, double pY)
     {
          x = pX;
          y = pY;
     }

}
```

Comme déjà indiqué, une méthode peut avoir plusieurs paramètres. Dans ce cas, les paramètres sont séparés par une virgule. Il faut indiquer pour chaque paramètre de quel type il s'agit. Dans le cas de la méthode **setCoordinates**, les deux paramètres sont des nombres réels (type **double**).

Exercice résolu :

- Ouvrez votre projet **00_Exercice_resolu** que vous avez créé auparavant.
- Ajoutez à la classe **Ball** la méthode **calculateVolume** qui calcule et retourne le volume (en cm^3) de la balle.
- Ajoutez un attribut **pressure** (nombre décimal, valeur initiale 0) à la classe **Ball.**
- Ajoutez une méthode **setPressure** à la classe, qui sert à modifier l'attribut **pressure**.
- Complétez la méthode **printInfo**.
- Créez une instance de la classe **Ball** et testez les nouvelles méthodes. (Consultez Internet pour trouver des valeurs réalistes pour les pressions des balles de tennis, volleyball, basketball,...)

Remarque avancée : Objets comme paramètres

Un paramètre peut avoir un type simple (**int**, **double**, **boolean**, …), mais un paramètre peut aussi être un objet (p.ex. du type **Rectangle**, **Point**, …).

Exemple :

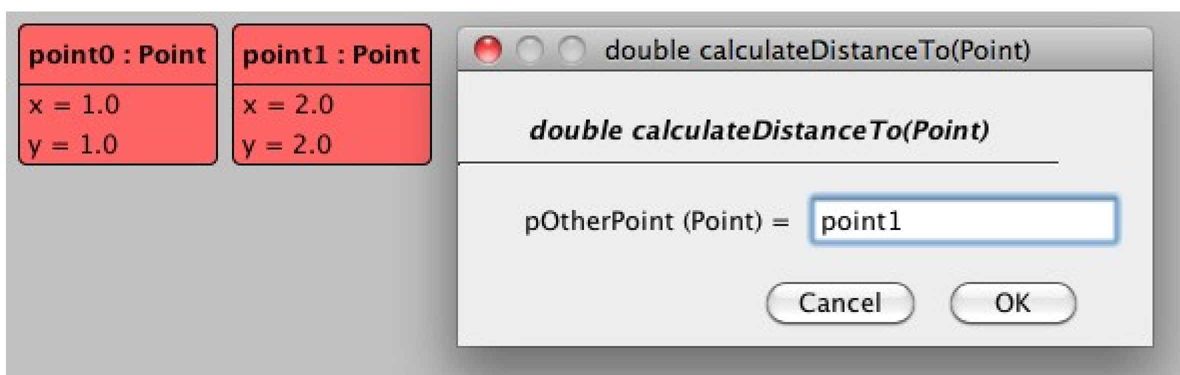
On pourrait par exemple imaginer une méthode **calculateDistanceTo** qui calcule la distance entre deux instances du type **Point** :

```
public class Point
{
	private double x = 0;
	private double y = 0;

	public void setCoordinates(double pX, double pY)
	{
		x = pX;
		y = pY;
	}

	public double calculateDistanceTo(Point pOtherPoint)
	{
		return Math.sqrt(
				Math.pow(pOtherPoint.x-x,2)+
				Math.pow(pOtherPoint.y-y,2));
	}
}
```

D'après le cours de mathématiques, la distance entre les points $A(x_A, y_A)$ et $B(x_B, y_B)$ se calcule suivant la formule : $AB=\sqrt{(x_B-x_A)^2+(y_B-y_A)^2}$

Pour les détails concernant la classe Math, veuillez vous référer au chapitre 1.13.

Pour tester la méthode, il faut créer deux objets **point0** et **point1** du type **Point**. Ensuite, on peut appeler la méthode **calculateDistanceTo** de **point0** et fournir comme paramètre le nom de la deuxième instance **point1**.



Dans cet exemple, la méthode va retourner la valeur 1.4142...

1.7. Qu'est-ce qu'un « type » ?

Un ordinateur sait traiter différentes sortes de données. En travaillant avec un tableur (Excel/Calc), vous avez certainement déjà remarqué qu'il faut faire la différence entre des données numériques et du texte. En plus, vous avez vu que l'ordinateur sait faire des opérations différentes avec du texte qu'avec des nombres et que le programme est très sensible si on essaie de mélanger différentes sortes de données.

En général, on parle de « type » ou de « type de donnée ». En effet, dans un langage de programmation évolué comme Java, toutes les données sont typisées, c.-à-d. le type de chaque donnée doit être fixe et connu à l'avance.

Le **type d'une donnée** définit :

- de quelle **sorte** de donnée il s'agit (nombre entier, nombre décimal, texte, …),
- dans quel **domaine** la valeur de la donnée peut varier,
- combien de **place** la donnée occupe dans la mémoire de l'ordinateur,
- quelles **opérations** peuvent être effectuées sur cette donnée.

En Java, on distingue les **types de base** suivants :

Les nombres entiers :

- **`byte`** 8 bits [-128 … 127]
- **`short`** 16 bits [-32'768 … 32'767]
- **`int`** 32 bits [-2'147'483'648 … 2'147'483'647]
- **`long`** 64 bits [-9'223'372'036'854'775'808 … 9'223'372'036'854'775'807]

Les nombres décimaux (valeurs négatives et positives) :

- **`float`** 32 bits [$1,4 \cdot 10^{-45}$ … $3,4 \cdot 10^{38}$]
- **`double`** 64 bits [$4,9 \cdot 10^{-324}$ … $1,79 \cdot 10^{308}$]

Les booléens :

- **`boolean`** 1 bit {true, false}

Remarques :

- La plupart du temps, nous allons employer le type **`int`** pour des entiers et le type **`double`** pour des nombres décimaux.
- Pour traiter des textes, on emploie le type **`String`** qui n'est pas un type de base et qui sera traité plus tard dans un chapitre à part. On peut cependant déjà noter que dans un programme Java les textes sont à noter entre guillemets : **`"…"`**
  (p.ex. : `streetName = "rue Bellevue";`)
  On peut concaténer (coller ensemble) des textes avec le symbole d'addition «**+**» .

Exemple : la classe `Rectangle`

```
public class Rectangle
{
     private int width  = 0;
     private int height = 0;

     public int getCircumference()
     {
          return 2 * (width+height);
     }

     public double calculateLengthOfDiagonal()
     {
          return Math.sqrt(width*width + height*height);
     }

     public String toString()
     {
          return "Rectangle, width:" + width + ", height:" + height;
     }

     public boolean isSquare()
     {
          return (width==height);
     }
}
```

Rectangle
− width : int
− height : int
+ getCircumference() : int
+ calculateLengthOfDiagonal() : double
+ toString() : String
+ isSquare() : boolean

Explications :

- La méthode **`getCircumference`** retournera un entier (type **`int`**).
- La méthode **`calculateLengthOfDiagonal`** utilise le théorème de Pythagore pour calculer la longueur de la diagonale. La racine carrée est calculée à l'aide de la méthode prédéfinie **`Math.sqrt`** (→ voir chapitre 1.13) qui peut prendre des entiers ou des décimaux comme paramètres, mais qui retourne toujours un nombre décimal comme résultat. Ainsi la valeur retournée par **`calculateLengthOfDiagonal`** doit être du type **`double`**.
- La méthode **`toString`** retourne une description textuelle de l'objet et aura donc un résultat du type **`String`.**

  Remarque :

  Habituellement toutes les classes en Java contiennent une méthode **`toString`** qui donne une description textuelle des objets.
- La méthode **`isSquare`** teste si le rectangle est carré ou non. La réponse à la question est soit *vrai* (**`true`**), soit *faux* (**`false`**), ce qui correspond au type de retour **`boolean`**. Le test[1] **`(width==height)`** fournit une valeur booléenne qui est directement retournée comme résultat.

1 Voir aussi chapitre 3.2 sur les opérateurs de comparaison

1.8. Conventions de noms pour les méthodes

Dans les exemples précédents, nous avons employé différents préfixes pour relever le rôle de différentes méthodes. Avec ces noms, nous avons suivi les conventions de noms qui existent pour les identifiants des méthodes en Java :

1.8.1. Les accesseurs - préfixe ***get***

Par le préfixe **`get`**, on désigne une méthode qui retourne directement la valeur d'un attribut. Une telle méthode est encore appelée un **accesseur**. Un accesseur a toujours un type retour différent de **`void`**. Le plus souvent, les accesseurs n'ont pas de paramètres. (Le préfixe **`get`** peut aussi être employé pour des méthodes qui retournent une valeur dérivée des attributs par un calcul très simple, même s'il ne s'agit pas accesseur dans le sens pur du terme[2]).

Exemples : `getRadius, getDiameter, getAge, getName, getX, getY`

1.8.2. Les manipulateurs - préfixe ***set***

Par le préfixe **`set`**, on désigne une méthode qui permet de modifier directement la valeur d'un ou de plusieurs attributs. Une telle méthode est encore appelée un **manipulateur**. Un manipulateur retourne en général **void** et il possède un ou plusieurs paramètres. Dans ce cours, nous supposons (sans vérification) que les données fournies correspondent toujours au domaine de définition de la classe. Si l'utilisateur fournit des données incorrectes, la classe se trouve dans un état inconsistant et les résultats sont imprévisibles.[3]

Exemples : `setDiameter, setTime`

1.8.3. Les méthodes effectuant un calcul - préfixe ***calculate***

Par le préfixe **`calculate`**, on désigne une méthode qui réalise un calcul plus complexe et retourne le résultat de ce calcul.

Exemples : `calculateSumOfDivisors, calculateGreatestCommonDivisor`

1.8.4. Les méthodes booléennes - préfixes ***is/has***

Pour rendre le code plus lisible, les méthodes et accesseurs qui retournent une valeur booléenne (et les variables de type booléen) portent de préférence les préfixes **`is`** ou **`has`**. Plus rarement on peut même trouver des préfixes comme **`can`**, **`was`**, **`allows`**, `...`

Exemple : `isSquare, isEven, isRunning, hasWheels, hasHitAWall, canFly`

1.8.5. La méthode ***toString***

En Java, en principe tous les objets peuvent être 'affichés' à l'aide de **`System.out.println`** ou concaténés avec un texte. Dans ces cas, Java appelle automatiquement la méthode **`toString`** de l'objet. Ainsi, tous les objets possèdent directement ou indirectement une méthode **`toString`** qui retourne les informations de base de l'objet sous forme d'un texte. Ces informations sont en général les valeurs actuelles des attributs principaux. La méthode **`toString`** n'a jamais de paramètres et elle retourne toujours une valeur du type **`String`**.

Exemple : `Rectangle.toString` (→ voir chapitre 1.7)

2 En effet, pour l'utilisateur des méthodes d'une classe il n'est pas nécessaire de savoir si c'est le rayon qui est mémorisé (et que le diamètre est dérivé du rayon) ou si c'est l'inverse...

3 Dans des programmes professionnels de telles erreurs sont évitées par le lancement et le traitement ***d'exceptions.*** Le concept d'exceptions ne fait pas partie du cours de la formation technique générale.

1.9. Qu'est-ce qu'un « constructeur » ?

Le « **constructeur** » est la méthode qui est appelée lors de la construction d'un objet. Il s'agit d'une méthode spéciale dont le rôle est d'**initialiser les attributs** et sous-objets de la classe afin de mettre l'objet dans son état initial.

Ce qu'il faut savoir à propos du constructeur :

- Tout constructeur porte toujours le **nom de la classe**.
- Un constructeur n'a **pas de type de retour**.
- Un constructeur peut avoir des **paramètres**.
- Un constructeur doit être **public** (**`public`**).
- La définition d'un constructeur n'est **pas obligatoire**.
- Une même classe peut posséder **plusieurs constructeurs**.
  Attention : Les différents constructeurs d'une classe ne peuvent pas avoir la même liste de paramètres : Ou bien le nombre de paramètres doit être différent ou bien au moins l'un des paramètres doit avoir un type différent. (Les noms des paramètres ne jouent pas de rôle pour la différenciation.)

Voici l'exemple de la classe **`Point`** définie précédemment :

Point
− x : double
− y : double
+ Point(pX : double, pY : double)
+ setCoordinates(pX : double, pY : double) : void

Le code source du **constructeur** est le suivant :

```
public class Point
{
     private double x;
     private double y;

     public Point(double pX, double pY)
     {
          x = pX;
          y = pY;
     }
}
```

Le compilateur Java reconnaît le constructeur au fait qu'il a le même nom que la classe et qu'il n'a pas de type retour.

Le **constructeur** **`Point(double pX, double pY)`** initialise le point lors de la création d'un objet du type **`Point`** et assure de ce fait que ses coordonnées soient toujours initialisées. L'initialisation des attributs se fait souvent dans le **constructeur**.

Remarque : Comme pour les manipulateurs (→ voir chap. 1.8.2), nous supposons dans ce cours (sans vérification) que les données fournies correspondent toujours au domaine de définition de la classe. Si l'utilisateur fournit des données incorrectes, l'objet se trouve dans un état inconsistant et les résultats sont imprévisibles.

1.10. Qu'est-ce qu'une « variable » ?

Une variable nous aide en général à mémoriser temporairement une donnée lors d'un calcul ou d'une autre opération à l'intérieur d'une méthode. Une variable est déclarée dans le corps d'une méthode et elle n'est accessible que dans ce bloc de cette méthode et à partir de la ligne dans laquelle elle est déclarée. On regroupe souvent toutes les déclarations de variables tout au début d'une méthode.

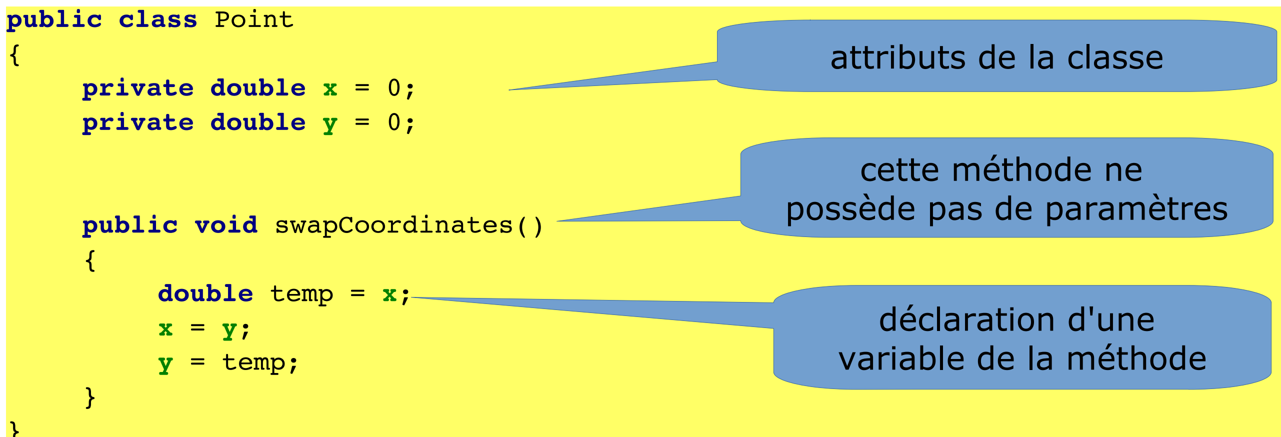
Résumons les **caractéristiques d'une variable** :

- une variable est d'un certain type,
- elle possède un nom,
- elle peut être initialisée lors de sa déclaration (comme un attribut),
- elle est connue uniquement dans le bloc de sa déclaration à l'intérieur d'une méthode.

Comparaison : Une variable est assez similaire à un paramètre ou un attribut, mais

- un **attribut** est une propriété fondamentale de la classe. Il est déclaré au début de la classe et il est accessible partout dans toutes les méthodes de la classe,
- un **paramètre** sert à fournir des données à une méthode lors de son appel. Il est déclaré dans l'en-tête de la méthode et n'est connu que dans cette méthode.
- une **variable** sert à mémoriser temporairement une valeur. Elle n'est connue que dans un domaine restreint de la méthode et son rôle dans le programme est très limité.

Voici l'exemple de la classe `Point` à laquelle on a rajouté la méthode `swapCoordinates`. Cette méthode échange les valeurs des coordonnées x et y :



```java
public class Point
{
     private double x = 0;
     private double y = 0;


     public void swapCoordinates()
     {
          double temp = x;
          x = y;
          y = temp;
     }
}
```


Explications et remarques :

- La variable `temp` est du type `double`, donc un nombre réel. Dans cet exemple, elle est initialisée lors de sa déclaration.
- Les trois lignes de code servent à échanger les valeurs des coordonnées à l'aide d'une variable temporaire.
- Comme pour les attributs, il est possible de déclarer une variable ***sans*** l'initialiser :

```java
double temp;
```

Reprenons la classe **Point** et ajoutons une méthode ayant besoin de paramètres :

Point
− x : double
− y : double
+ Point(pX : double, pY : double)
+ setCoordinates(pX : double, pY : double) : void
+ swapCoordinates() : void
+ move(pDeltaX : double, pDeltaY : double) : void

```
public class Point
{
     private double x = 0;
     private double y = 0;

     public Point(double pX, double pY)
     {
          x = pX;
          y = pY;
     }

     public void setCoordinates(double pX, double pY)
     {
          x = pX;
          y = pY;
     }

     public void swapCoordinates()
     {
          double temp = x;
          x = y;
          y = temp;
     }

     public void move(double pDeltaX, double pDeltaY)
     {
          x = x + pDeltaX;
          y = y + pDeltaY;
     }
}
```

attributs de la classe

à partir d'ici, cette méthode possède une variable “temp”

à partir d'ici, la variable “temp” n'existe plus

cette méthode possède les deux paramètres “pDeltaX” et “pDeltaY” et change les attributs “x” et “y”

Explications :

- La méthode **move** ajoute des valeurs aux attributs afin de déplacer le point.
- Veuillez noter que sans explications, le code n'est pas très compréhensible. Afin d'éviter une telle situation, on préfère ajouter des **commentaires** au code source (→ voir chapitre 2.1).

1.11. Opérateurs, compatibilité et conversions

Dans les exemples précédents, vous avez déjà retrouvé les opérateurs standards qui vous sont familiers (+ , - , * , /). Complétons la liste et donnons quelques précisions.

1.11.1. Opérateur d'affectation « = »

Exemple :

```
seconds = minutes*60 + hours*3600;
```

Effet : La partie qui se trouve à droite du signe d'égalité est évaluée et le résultat est affecté (mémorisé) dans la variable (ou l'attribut) qui se trouve à gauche du signe d'égalité.

Il faut que les types des deux parties soient **compatibles**, c.-à-d que leurs types doivent être égaux ou que le type du résultat à droite doit être un type 'plus petit' que le type à gauche.

P. ex. on peut affecter simplement une valeur du type **`int`** à une variable du type **`long`** ou **`double`**, mais pas inversement. Sinon il faut forcer la conversion (→ voir chapitre 1.11.3) vers un type plus petit.

Exemples :

```
double x = 123.3;
int i = 102;
i = x;          // impossible !!
i = (int)x;     // conversion forcée avec perte de précision : i sera 123
x = i;          // pas de problèmes !
```

Remarque pour avancés :

En Java, il existe des opérateurs d'affectation qui effectuent en même temps un calcul (**++** , **+=** , ***=** , …). Il n'y a aucune nécessité de les utiliser dans ce cours, mais si cela vous intéresse, vous trouvez des précisions dans tous les livres ou sites sur Java.

1.11.2. Opérateurs arithmétiques

+	addition
-	soustraction
*****	multiplication
/	division (entière et rationnelle !)
%	modulo (reste d'une division entière)

Exemples:

2.5 / 2.0	→	1.25	division de réels → résultat réel
20 / 7	→	2	division d'entiers → résultat entier
20 % 7	→	6	le reste de la division de 20 par 7 donne 6

En effet : $20 = \underbrace{2}_{\text{partie entière}} \cdot 7 + \underbrace{6}_{\text{reste}}$

1.11.3. Conversions forcées de types (explicites)

En faisant précéder une donnée par le nom d'un autre type entre parenthèses, la donnée est convertie dans le type entre parenthèses. On appelle ceci **conversion forcée** ou aussi **conversion explicite**.

Ceci peut être utile par exemple si on veut tronquer la partie décimale d'un nombre réel :

(int) 3.75 → 3

A cause de la compatibilité des types (→ voir chapitre 1.11.1), il n'est en général pas nécessaire de convertir un type plus petit (p.ex. **int**) en un type plus large (p.ex. **double**). Une telle conversion fait du sens dans le cas, où on veut forcer une division réelle avec des données entières (voir dernier exemple de cette page).

1.11.4. Conversions automatiques de types (implicites)

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun. On appelle ceci **conversion automatique** ou aussi **conversion implicite**.

En principe des types plus 'petits' sont convertis en des types plus 'larges'; de cette façon on ne perd pas en précision.

Exemples :

2 + 2.5	→ 4.5	un entier et un réel	→	résultat réel
3 / 2	→ 1	deux entiers	→	division entière avec résultat entier
3 / 2.0	→ 1.5	un entier et un réel	→	division réelle avec résultat réel

Ce dernier exemple montre bien comment on peut obtenir une division réelle pour des données du type entier.

Autre exemple:

```
double middle;
int x1 = 20;
int x2 = 15;
middle = (x1+x2) / 2.0;        // résultat réel : 17.5
```

Attention :

Dans des calculs plus complexes, il faut tenir compte de l'ordre dans lequel les opérations sont évaluées (→ **priorité des opérateurs** mathématiques) :

5 + 10 / 4	→ 7	division entière	puis addition d'entiers
5.0 + 10 / 4	→ 7.00	division entière	puis addition de réels
5 + 10.0 / 4	→ 7.50	division réelle	puis addition de réels
5.0 + 10.0 / 4	→ 7.50	division réelle	puis addition de réels

```
double x;
int i = 101;
x = i / 4;           // pas de conversion,      x aura la valeur 25.00
x = i / 4.0;         // conversion implicite,   x aura la valeur 25.25
x = (double)i / 4;   // conversion explicite,   x aura la valeur 25.25
```

Exercice de récapitulation :

Soient les déclarations :

```
int    n = 300;
int    m = 1000;
double r = 400.0;
double q = 5.35;
```

Indiquez le type et le résultat des expressions suivantes :

Expression	Type	Résultat
`m / 300.0`		
`r + m/n`		
`(r+n) / m`		
`r + n / m`		
`m % n`		
`(int)(q * 4)`		
`(int)q * 4.0`		
`r + m/(double)n`		

1.11.5. Opérateur de concaténation « + »

Deux textes (type **String**) peuvent être 'collés ensemble' à l'aide de l'opérateur de concaténation « **+** ». Lorsque Java détecte que le résultat est du type **String**, les données d'autres types (**int**, **double**, …) qui sont concaténées avec '+' sont automatiquement converties en texte (→ voir exemples pour **System.out.println** ci-dessous).

Exemple :

```
String noun = "Paul";
String verb = "aime";
String sentence = noun + " " + verb + " les bananes.";
```

1.12. Affichage d'une ligne de texte

L'instruction

```
System.out.println( … );
```

permet d'afficher un texte dans la fenêtre des messages et de passer à la ligne après l'affichage. L'utilité de la fenêtre des messages est assez restreinte, mais parfois elle nous permet de visualiser plus de détails que l'environnement de développement. Ainsi, nous pouvons par exemple utiliser cette instruction pour faire afficher des résultats intermédiaires lorsque nous sommes à la recherche d'une erreur dans notre programme.

Evidemment, on peut aussi afficher le **contenu de variables texte** (type **String**) avec une instruction d'affichage.

Comme Java exécute des conversions automatiques, on peut aussi afficher le **contenu de variables numériques** à l'aide de cette instruction. En combinaison avec l'opérateur de concaténation, cette instruction peut donc servir à afficher des messages assez complexes.

L'instruction

```
System.out.print( … );
```

permet d'afficher un texte *sans* passer à la ligne après l'affichage. Ainsi le *prochain* affichage continuera dans la même ligne.

Exemples :

```
int sideLength = 220;
System.out.println( "Hello" );      // affiche le texte Hello
System.out.println( sideLength );  // affiche le nombre 220
System.out.println( "Un côté du carré mesure " + sideLength + " cm" );
                        // affiche le texte : Un côté du carré mesure 220 cm
int surface = sideLength*sideLength;
System.out.println( "Surface du carré : " + surface + " cm2" );
                          // affiche le texte : Surface du carré : 48400 cm2
```

On peut intégrer des **appels de méthodes** dans une instruction d'affichage:

```
System.out.println("Temps actuel en secondes : " + getTimeInSeconds());
```

On peut intégrer des **calculs** dans une instruction d'affichage, mais alors il est recommandé de comprendre chaque calcul entre **parenthèses** (sinon Java risque de mal interpréter certains opérateurs) :

```
System.out.println("Surface du carré : "+(sideLength*sideLength)+" cm2");
System.out.println("Périmètre du rectangle : "+(2*(x2-x1)+2*(y2-y1))+"cm");
```

1.13. La classe « Math »

Dans la classe **Math** qui est contenue en Java, sont définies des constantes et des méthodes très utiles que nous pouvons employer dans nos programmes :

Math.PI	approximation très précise de Pi (**double**) Remarque : PI est une constante et n'est donc pas suivi de parenthèses. Les noms des constantes en JAVA sont écrites en majuscules.
Math.abs(...)	retourne la valeur absolue du nombre fourni comme paramètre
Math.max(... , ...)	retourne la plus grande des valeurs fournies comme paramètres
Math.min(... , ...)	retourne la plus petite des valeurs fournies comme paramètres
Math.round(...)	retourne la valeur entière la plus proche du nombre décimal fourni comme paramètre (**double** → **long** ; **float** → **int**)
Math.sqrt(...)	retourne la racine carrée du nombre fourni comme paramètre (**double**)
Math.pow(... , ...)	retourne X^Y pour deux décimaux fournis comme paramètres
Math.random()	retourne un nombre aléatoire positif dans le domaine [0.0 … 1.0[

```
surface = Math.PI * Math.pow(radius,2);
```

Autres méthodes intéressantes de la classe **Math** :

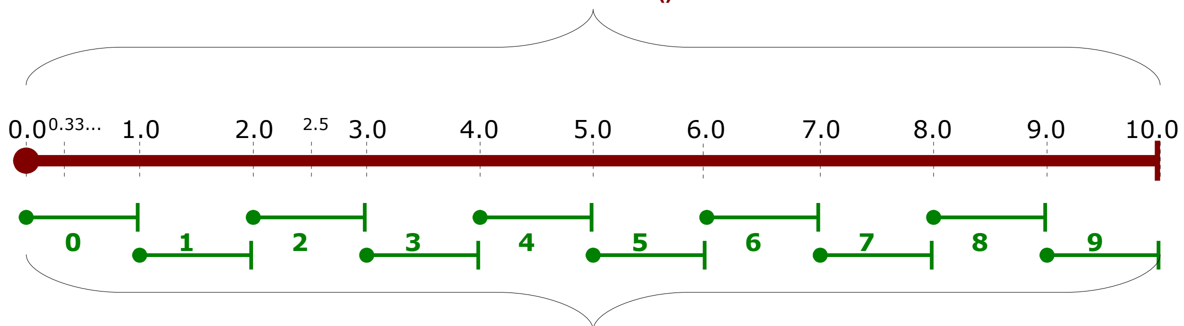
floor, sin, cos, tan, acos, asin, atan, exp, log, log10, …

1.14. *Générer des nombres aléatoires*

Chaque appel de la méthode **`Math.random()`** retourne un nombre réel aléatoire appartenant à l'intervalle [0.0 … 1.0[. Cependant nous avons souvent besoin de nombres aléatoires dans d'autres domaines ou nous voulons travailler avec des nombres entiers aléatoires.

Comment pouvons-nous générer des nombres réels aléatoires dans d'autres intervalles ?

Génération de nombres aléatoires réels	
Expression	**Résultat** un nombre **réel** aléatoire appartenant au domaine :
`Math.random()`	[0.0 … 1.0[
`Math.random() * 100`	[0.0 … 100.0[
`Math.random() * 100 - 50`	[-50.0 … 50.0[
Formule générale à retenir : **`Math.random() * (max - min) + min`**	**[min … max[**

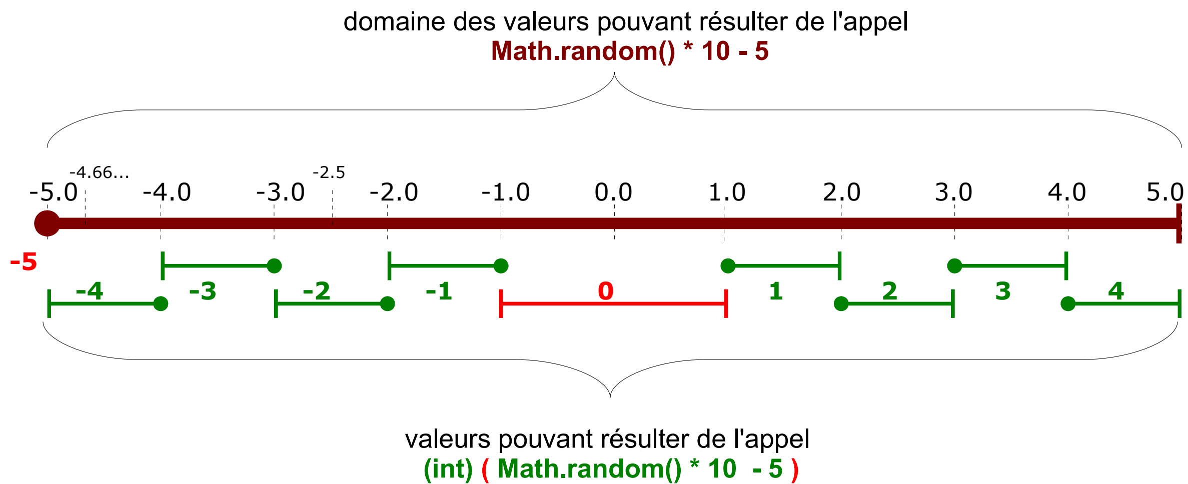


Génération de nombres aléatoires entiers	
Expression	**Résultat** un nombre **entier** aléatoire appartenant au domaine :
`(int)(Math.random() * 10)`	[0 … 9]
`(int)(Math.random() * 10) + 1`	[1 … 10]
`(int)(Math.random() * 10) - 5`	[-5 … 4]
`(int)(Math.random() * 11) - 5`	[-5 … 5]
Formule générale à retenir : **`(int) (Math.random() * (max-min+1)) + min`**	**[min … max]**

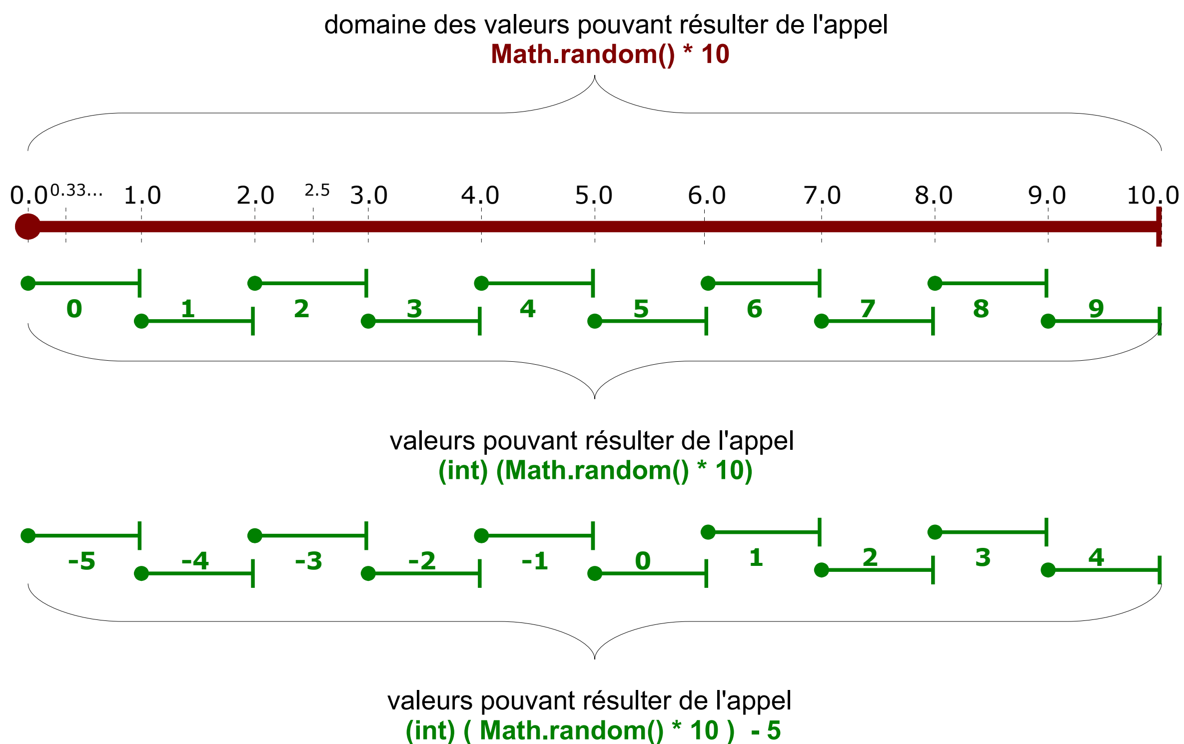
Attention :

Pour garantir que les valeurs entières soient réparties de façon régulière sur tout l'intervalle, il faut veiller à bien placer les parenthèses lors du casting. Considérez l'exemple suivant :

Parenthèses mal placées : (int) (Math.random() * 10 - 5) **VERSION INCORRECTE!**



Parenthèses bien placées : (int) (Math.random() * 10) - 5 **VERSION CORRECTE!**



Comme la soustraction est effectuée APRÈS le casting, les valeurs sont réparties de façon régulière sur l'intervalle!

2. Présentation et documentation du code

2.1. *Qu'est-ce qu'un « commentaire » ?*

Écrire une bonne documentation constitue un complément important à un code source de bonne qualité. La documentation permet de communiquer ses intentions aux lecteurs, d'obtenir une vue d'ensemble en langage naturel, plutôt que de les obliger à décrypter un code ardu. De plus les commentaires nous aideront à mieux nous retrouver dans notre code lorsque nous y retravaillerons plus tard. Un commentaire dans un code source est un texte qui n'est pas considéré par le compilateur. Il n'influence donc pas le déroulement du programme. Il existe trois types de commentaires :

- Un **commentaire uni-ligne** commence par les deux symboles consécutifs « **//** ». Tout ce qui suit ces deux symboles dans une ligne est ignoré par le compilateur (voir exemples précédents dans ce cours). Ce type de commentaires est en général utilisé pour expliquer l'utilité ou le fonctionnement d'un bloc d'instructions dans une méthode.
- Un **commentaire multi-ligne** commence par les deux symboles « **/*** » et se termine par les deux symboles « ***/** ». Tout ce qui se trouve entre le début et la fin est ignoré par le compilateur.
- Un **commentaire du type « JavaDoc »** commence par les trois symboles « **/**** » et se termine par les deux symboles « ***/** ». Tout ce qui se trouve entre le début et la fin est ignoré par le compilateur mais pris en charge par le **générateur de documentation automatique** :
  Les commentaires « JavaDoc » ont une double finalité, parce qu'ils servent aussi à générer automatiquement un fichier de documentation pour la classe et pour tous les éléments qui y sont définis. Nous allons profiter de cette fonctionnalité dans Unimozer (et NetBeans).

2.1.1. Règles pour l'utilisation des commentaires «JavaDoc»

La description principale d'une **classe** doit indiquer son objectif en termes généraux.

```
/**
 * Cette classe calcule le poids normal d'après Broca et l'indice de masse
 * corporelle (BMI) pour un homme ou une femme
 */
public class BodyStatistics
```

Cette description doit rester assez générale, sans donner trop de détails sur son implantation. Bien souvent, une seule phrase suffira.
La description principale d'un **attribut** donne des détails concernant le but ou l'interprétation du contenu de l'attribut.

```
/**
 * Le sexe de la personne ("m" -> masculin,  "f" -> féminin)
 */
private String gender = "m";
```

La description principale d'une **méthode** explique le but de la méthode.

```
/**
 * Méthode qui permet de définir l'âge de la personne
 */
public void setAge(int pAge)
```

2.1.2. Précisions : les commentaires « JavaDoc » des classes

Dans les commentaires des classes, les balises **@author** et **@version** sont prises en compte par le générateur de documentation automatique. Leur signification est la suivante :

@author le nom de l'auteur de la classe

@version la version de la classe (contient souvent la date de la dernière modification)

```
/**
 * Cette classe calcule le poids normal d'après Broca et l'indice de masse
 * corporelle (BMI) pour un homme ou une femme
 *
 * @author     J. Valjean
 * @version    21/01/2011
 */
```

2.1.3. Précisions : les commentaires « JavaDoc » des méthodes

Dans les commentaires des méthodes, les balises **@param** et **@return** sont prises en compte par le générateur de documentation automatique. Leur signification est la suivante :

@param le nom et la description qui suivent sont ceux d'un paramètre

@return description du résultat fourni par la méthode

D'après les conventions de Java il est obligatoire d'indiquer les marques **@param** et **@return** pour les méthodes. Les paramètres doivent être décrits dans le même ordre que dans l'en-tête de la méthode. **@return** est omis s'il s'agit d'une méthode retournant **void**.

La balise **@param** est utilisée avec des méthodes et des constructeurs :

```
/**
 * Méthode qui permet de définir l'âge
 * @param pAge     l'âge en années
 */
public void setAge(int pAge)
```

La balise **@return** ne sert que pour les méthodes :

```
/**
 * Méthode pour calculer le poids normal suivant la
 * formule de Broca
 * @return     poids normal en "kg"
 */
public double getNormalWeight()
```

Pour de plus amples informations concernant **« JavaDoc »**, veuillez consulter la page Internet suivante :

http://www.oracle.com/technetwork/java/javase/documentation/writingdoccomments-137785.html

2.2. *Qu'est-ce que « l'indentation » ?*

Par **indentation**, en allemand « Einrückung », on entend le fait de déplacer certaines parties du code source plus vers la droite que d'autres. Généralement les instructions contenues dans un bloc d'instructions sont **indentées d'une tabulation**, c'est-à-dire qu'il y a une tabulation en plus devant chaque ligne par rapport à celles au niveau du bloc d'instructions correspondant ('bloc d'instructions' → voir aussi chapitre 3.7).

Voici une partie de la classe **Point** qui nous sert d'exemple :

```
public class Point
{
    private double x = 0;
    private double y = 0;

    /**
     * Déplace le point dans le plan mathématique
     * @param  pDeltaX       le déplacement de l'abscisse
     * @param  pDeltaY       le déplacement de l'ordonnée
     */
    public void move(double pDeltaX, double pDeltaY)
    {
        /*
         Déplace le point dans le plan
        */
        x = x + pDeltaX;
        y = y + pDeltaY;
    }
}
```

Niveau d'indentation 2 : Le code source est indenté d'une tabulation par rapport au niveau 1 et de deux tabulations par rapport au niveau 0.

Niveau d'indentation 1 : Le code source est indenté d'une tabulation par rapport à la ligne précédente.

Niveau d'indentation 0 : Le code source colle à la bordure gauche.

3. Les structures de contrôle

Les programmes développés jusqu'à présent se composent d'instructions qui sont exécutées de manière **séquentielle** : les instructions sont exécutées du haut vers le bas. Cependant, afin de pouvoir écrire des programmes plus utiles, il est indispensable de disposer de structures de contrôle qui peuvent **changer la suite de l'exécution** des instructions.

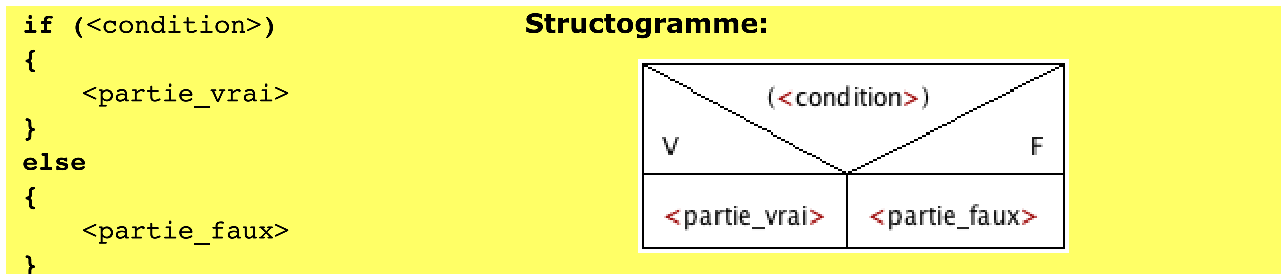
Ces structures permettent d'une part d'influencer le déroulement du programme et d'autre part de rendre plus simple certaines parties du code source. A partir de ce chapitre, nous allons employer de façon régulière des **structogrammes** (angl. : *NSD - Nassi-Shneiderman Diagrams*) pour représenter la structure de nos méthodes de façon graphique.

3.1. La structure alternative

La **structure alternative**, encore appelée structure « **if** », permet de faire un **choix** entre deux actions suivant qu'une certaine condition soit remplie ou non.

Voici la syntaxe de la **structure alternative** en Java et sous forme de structogramme :

```
if (<condition>)
{
    <partie_vrai>
}
else
{
    <partie_faux>
}
```



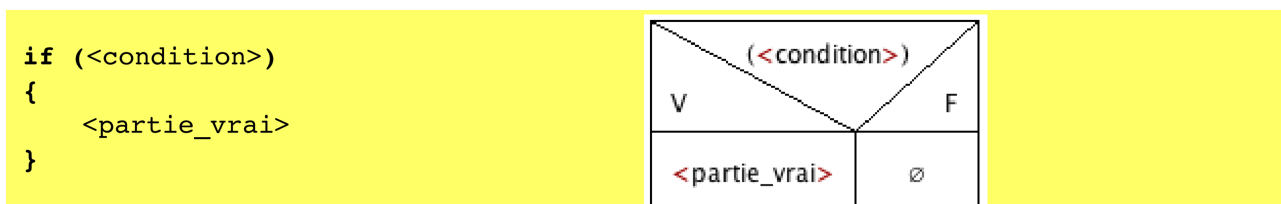
Les différents blocs d'instructions sont délimités par des accolades (comme le début et la fin d'une méthode). Pour les détails sur les blocs d'instructions → voir chapitre 3.7.

Si l'évaluation de la **<condition>** donne le résultat **vrai** (**true**), le bloc **<partie_vrai>** est exécuté et le bloc **<partie_faux>** est ignoré.

Si l'évaluation de la **<condition>** donne le résultat **faux** (**false**), le bloc **<partie_faux>** est exécuté et le bloc **<partie_vrai>** est ignoré.

S'il n'existe pas de **<partie_faux>**, le bloc « **else** » peut simplement être omis :

```
if (<condition>)
{
    <partie_vrai>
}
```



Si **<partie_vrai>** ou **<partie_faux>** ne contiennent qu'une seule instruction, il n'est pas nécessaire de noter les accolades :

```
if (<condition>) <partie_vrai>;
else  <partie_faux>;
```

respectivement :

```
if (<condition>) <partie_vrai>;
```

Exemples :

Calculer le maximum de deux valeurs

```
if (a>b)
{
    max = a;
}
else
{
    max = b;
}
```

ou aussi :

```
if (a>b) max = a;
else     max = b;
```

Garantir que **a** contienne la plus grande de deux valeurs réelles et **b** la plus petite (→ permutation)

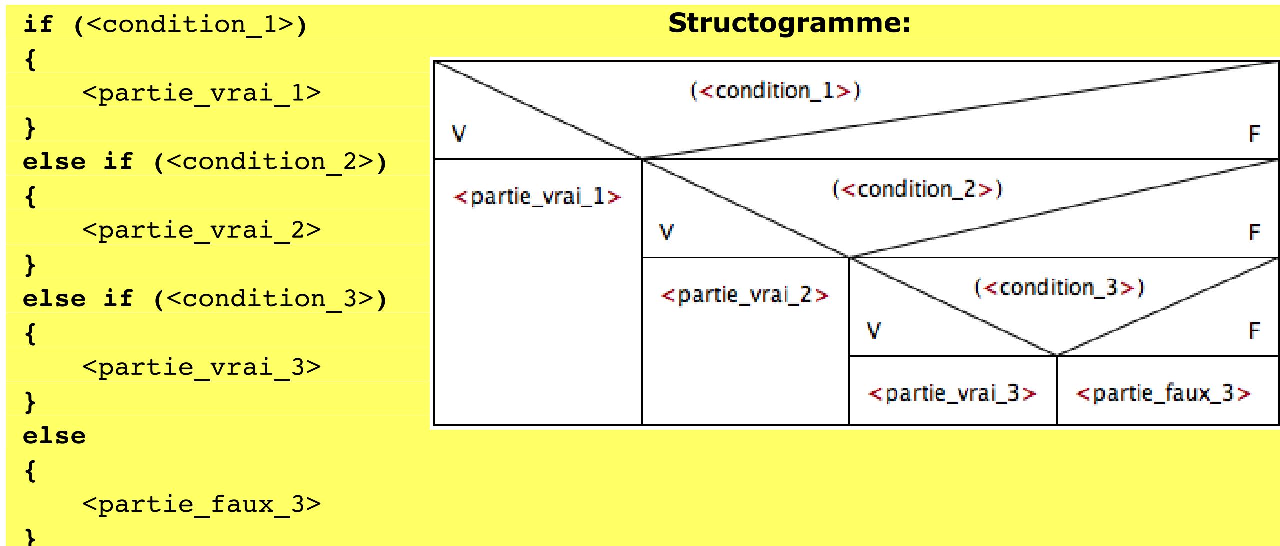
```
if (a<b)
{
    double help = a;
    a = b;
    b = help;
}
```

Remarques:

- Puisqu'une condition peut avoir la valeur ***true*** ou ***false***, on peut parler **d'expression logique ou expression booléenne** au lieu de condition. La <condition> peut donc être une simple variable booléenne, un appel d'une méthode retournant le type **boolean**, ou toute autre expression retournant le type **boolean**.
- Il n'y a jamais de point-virgule '**;**' derrière une accolade fermante '**}**' d'un bloc d'instructions.
- Il n'y a pas de point-virgule '**;**' derrière la parenthèse fermante '**)**' de la condition. En effet, dans une construction de la forme **if(…);{…}** le point-virgule est interprété comme une instruction vide et la structure **if** se rapporterait uniquement à cette instruction vide. Le bloc d'instruction qui suit ne serait plus dépendant de la condition et serait TOUJOURS exécuté...
- Pour combiner plusieurs conditions, on peut **imbriquer** (all: *verschachteln*) les conditions:

```
if (<condition_1>)
{
    <partie_vrai_1>
}
else if (<condition_2>)
{
    <partie_vrai_2>
}
else if (<condition_3>)
{
    <partie_vrai_3>
}
else
{
    <partie_faux_3>
}
```

Structogramme:



3.2. La structure alternative et le retour de résultats

Dans une méthode qui retourne un résultat (autre que **void**), il faut veiller à **retourner un résultat dans TOUS les cas de la méthode**, sinon le compilateur produit une erreur du type *'missing return statement'*.

Exemples :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
}                                    <== missing return statement
```

Cette méthode ne se laisse pas compiler et elle produit une erreur *'missing return statement'*, car elle ne retourne pas de valeur si la condition n'est pas remplie.

Même l'exemple suivant produit une erreur :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
    if (pA<=pB)
        return pB - pA;
}                                    <== missing return statement
```

En analysant le code, nous voyons bien que les deux conditions traitent tous les cas possibles, mais le compilateur traite les deux structures **if** séparément et ne fait pas ce rapprochement.

Solutions :

1. Dans des cas simples (comme dans cet exemple), nous pouvons employer une seule instruction **if** avec un bloc **else** :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
    else
        return pB - pA;
}
```

2. Dans des cas plus complexes, il est recommandable d'employer **une seule instruction return à la fin de la méthode** et une **variable intermédiaire** pour la valeur du résultat :

```
public int test(int pA, int pB)
{
    int result = 0;
    if (pA>pB)
        result = pA - pB;
    else
        result = pB - pA;
    return result;          // <== une seule instruction return qui est
                            //   toujours exécutée à la fin de la méthode
}
```

3.3. Les opérateurs de comparaison

Pour pouvoir formuler des conditions simples, on peut se servir des opérateurs de comparaisons suivants :

`==`	est égal à
`!=`	est différent de
`>`	est strictement supérieur à
`>=`	est supérieur ou égal à
`<`	est strictement inférieur à
`<=`	est inférieur ou égal à

Le résultat d'une comparaison est toujours une valeur du type booléen.

Exemples :

Si nous partons des déclarations suivantes,

```
double  price;
int     a,b,c,divisor;
boolean found;
```

nous pouvons formuler les conditions suivantes :

`price>1000`	test, si la valeur de la variable **price** est supérieure à 1000
`a==b`	test, si la valeur de **a** est égale à la valeur de **b**
`divisor!=0`	test, si la valeur de **divisor** est différente de 0
`found`	test, si la variable **found** a la valeur **true**

Attention :

Les opérateurs de comparaison fonctionnent uniquement pour des types de base (**int**, **float**, **double**, **char**, ...). Pour des objets et des variables du type **String**, il faut utiliser des méthodes spéciales.

3.4. Les opérateurs logiques

Les conditions peuvent être combinées à l'aide de la **conjonction** (**ET** logique) et de la **disjonction** (**OU** logique). On peut inverser logiquement le résultat d'une condition à l'aide de la **négation** (**NON** logique).

opérateur	**opération logique**	**désignation**	**retourne *true*, si et seulement si ...**
&&	ET	conjonction	... les deux conditions sont remplies
||	OU	disjonction	... au moins l'une des deux conditions est remplie
!	NON	négation	... la condition derrière ! retourne `false`

Soient C1 et C2 deux conditions (ou deux expressions booléennes) :

C1	**C2**	**C1 && C2**	**C1 || C2**	**!C1**
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Exemples de conditions composées :

`(x>0) && (x<=10)`	test, si x appartient à l'intervalle]0,10], donc si **x** est strictement plus grand que 0 **et** **x** est inférieur ou égal à 10
`!(a==b)`	test, si **a** n'est pas égal à **b** (même effet que `a!=b`)
`(a==0) || (b==0)`	test, si au moins l'une des deux variables **a** ou **b** est 0, donc si **a** est 0 **ou** **b** est 0 **ou** **a** et **b** sont 0
`found && !list.isEmpty()`	test, si la variable **`found`** est **`true`** **et** la méthode **`list.isEmpty()`** retourne **`false`**

Attention - La loi de De Morgan -

Soient C1 et C2 deux conditions (ou deux expressions booléennes), alors :

`!( C1 && C2)`	est identique à	`!C1 || !C2`
`!( C1 || C2)`	est identique à	`!C1 && !C2`

Exemple illustratif :

Papa tond la pelouse,	s'il fait beau	**ET**	la pelouse est trop haute.
Papa **ne** tond **pas** la pelouse,	s'il fait mauvais	**OU**	la pelouse est courte.

Exemples :

`!((a==0) || (b==0))`	est identique à	`(a!=0) && (b!=0)`
`!((a>b) && (b>c))`	est identique à	`(a<=b) || (b<=c)`

3.5. La structure répétitive « tant que »

La **structure répétitive « tant que »**, encore appelée structure **« while »**, permet de répéter certaines instructions tant qu'une condition donnée est vraie.

La syntaxe de la **structure répétitive « tant que »** est la suivante :

```
while (<condition>)
{
    <instructions>
}
```

- Tant que la **<condition>** est vraie, les **<instructions>** sont exécutées. Dès que la **<condition>** est fausse, la boucle s'arrête.
- Si la **<condition>** est fausse dès le début, le corps de la boucle n'est jamais exécuté.
- Une boucle dont la **<condition>** est toujours vraie ne s'arrête jamais. On parle alors d'une « boucle infinie ». Bien entendu, il faut éviter une telle boucle, puisqu'elle fait en sorte que le programme bloque et déstabilise tout l'ordinateur.

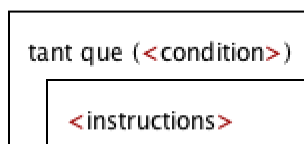
Si le corps de la **structure répétitive « tant que »** ne contient qu'une seule instruction, on peut omettre les accolades :

```
while (<condition>)
      <instruction>;
```

Remarques :

- Il n'y a jamais de point-virgule '**;**' derrière une accolade fermante '**}**' d'un bloc d'instructions.
- Il n'y a pas de point-virgule '**;**' derrière la parenthèse fermante '**)**' de la condition. En effet, dans une construction de la forme **while(…); {…}** le point-virgule serait interprété comme une instruction vide et la structure **while** se rapporterait uniquement à cette instruction vide. Comme une instruction vide n'effectue pas de changement dans l'état de la condition, une fois que la condition a été remplie, elle reste remplie et la boucle est infinie. (Le bloc d'instructions qui suit n'est pas lié à la boucle et ne sera jamais atteint si la condition est **true**. Si la condition est **false**, la boucle n'aura pas d'effet, mais le bloc d'instructions sera exécuté une seule fois.).

La représentation d'une **structure répétitive « tant que »** dans un structogramme est la suivante :



Exemple :[4]

Voici l'exemple d'une boucle qui recherche le plus grand commun diviseur de deux nombres (angl.: ***GCD*** *Greatest Common Divisor*) :

```java
int a = 28;
int b = 21;


// déterminer le minimum
int gcd=a;
if (b<gcd) gcd=b;


while( (a % gcd !=0) || (b % gcd !=0) )
{
   gcd=gcd-1;
}

System.out.println("GCD = " + gcd);
```

```
a ← 28
b ← 21
gcd ← a
(b < gcd)
V: gcd ← b | F: ∅
tant que ( (a % gcd != 0) || (b % gcd != 0) )
    gcd ← gcd – 1
écrire gcd
```

Remarque pratique :

Dans ce cours nous allons éviter de quitter une boucle **while** par un **return** dans le bloc d'instructions de la boucle. Pour terminer une boucle, nous allons définir une condition de parcours appropriée. De cette façon, le code reste plus clair car il ne faut pas analyser le corps entier de la boucle pour être sûr qu'il n'y a pas "d'issues cachées" dans la boucle. En plus on évite des erreurs du type *'missing return statement'*.

4 Site vous aidant à visualiser les passages d'une boucle : https://www.cs.virginia.edu/~lat7h/cs1/JavaVis.html

3.6. La structure répétitive « pour »

La **structure répétitive « pour »**, encore appelée boucle **« for »** ou **boucle de comptage** (all. : « *Zählschleife* »), permet de répéter un bloc d'instructions un nombre précis de fois. En effet, le nombre d'itérations (de répétitions) est **connu à l'avance**.

Exemple :

Pour commencer, voici une simple boucle **for** qui affiche 10 fois "Hello" à l'écran.

```
for (int i=0 ; i<=9 ; i++)
{
   System.out.println("Hello");
}
```

pour i ← 0 à 9

écrire "Hello"

Voici la syntaxe d'une **structure répétitive « pour »** en Java :

```
for ( <initialisation> ; <condition> ; <incrémentation> )
{
    <instructions>
}
```

- Dans la partie **<initialisation>** le compteur (p.ex. une variable i) est initialisé. Souvent cette variable est même déclarée dans cette partie (voir exemple).
- La condition **<condition>** est évaluée *avant* chaque itération. Si elle est vraie, une nouvelle itération est faite, sinon la boucle se termine. Typiquement dans une boucle de comptage, la condition a la forme :
  i ≤ valeur_maximale ou bien **i ≥ valeur_minimale**
- Le code dans **<incrémentation>** est exécuté *après* chaque itération. Dans ce cours, nous utilisons ou bien l'incrémentation positive (**i++** ou **i=i+1**) ou bien l'incrémentation négative (**i--** ou **i=i-1**), c.-à-d. à chaque passage de la boucle, le compteur est ou bien augmenté de 1 ou bien diminué de 1.
- Le bloc **<instructions>** forme le corps de la boucle et est exécuté à chaque répétition.

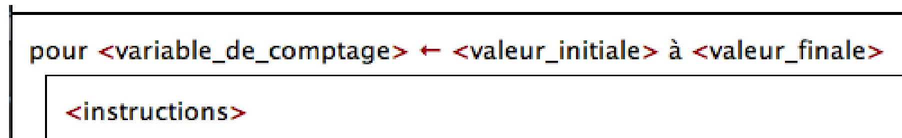
Si le corps de la **structure répétitive « pour »** ne contient qu'une seule instruction, on peut omettre les accolades :

```
for ( <initialisation> ; <condition> ; <incrementation> )
      <instruction>;
```

Remarques :

- Il n'y a jamais de point-virgule '**;**' derrière une accolade fermante '**}**' d'un bloc d'instructions.
- Il n'y a pas de point-virgule '**;**' derrière la parenthèse fermante '**)**' de l'instruction **for**. En effet, dans une construction de la forme **for(…); {…}** le point-virgule serait interprété comme une instruction vide et la structure **for** se rapporterait uniquement à cette instruction vide. Ainsi cette instruction vide serait 'exécutée' le nombre de fois fixé par la boucle **for**. Ensuite le bloc d'instructions qui suit serait exécuté une seule fois.

Représentation d'une **structure répétitive « pour »** dans un structogramme :



En comparant avec le code source ...

- c'est dans la partie **`<initialisation>`** que la **`<valeur_initiale>`** est affectée à la **`<variable_de_comptage>`**.
- c'est dans la partie **`<condition>`** que la valeur de la **`<variable_de_comptage>`** est comparée à la **`<valeur_finale>`**.
- c'est dans la partie **`<incrémentation>`** que la **`<variable_de_comptage>`** est augmentée ou diminuée. Dans le structogramme nous pouvons optionnellement indiquer le pas de l'incrémentation (**`pas=1`** ou **`pas=-1`**). Par défaut (c.-à-d. si aucun pas n'est indiqué), nous utilisons l'incrémentation positive (**`i++`**).

Remarques pratiques :

- Il ne faut jamais modifier la variable de comptage dans le corps de la boucle. Nous n'allons pas non plus modifier les valeurs initiales et finales de la boucle dans le corps de la boucle.
- Dans ce cours nous allons éviter de quitter une boucle **for** par un **return** dans le bloc d'instructions de la boucle. Si nous devons terminer une boucle de comptage avant qu'elle ne soit arrivée à la valeur finale, nous allons utiliser une boucle **while** avec une condition de parcours appropriée.
- En respectant les deux règles ci-dessus, il suffira toujours de regarder l'en-tête de la boucle **for** pour savoir exactement combien de fois elle sera parcourue. De cette façon, la boucle **for** garde son avantage principal : une bonne lisibilité et une interprétation facile. En plus on évite des erreurs du type *'missing return statement'*.

3.7. Les blocs et la durée de vie des variables

Dans ce chapitre, nous avons traité différentes structures de contrôle et dans ce contexte, nous avons introduit la notion de bloc d'instructions.

Un bloc d'instructions regroupe une suite d'instructions en les plaçant entre accolades { ... } .
Un bloc d'instructions peut être utilisé partout où on pourrait placer une seule instruction.

Les blocs servent à marquer le corps d'une méthode mais aussi à regrouper des instructions derrière **if**, **else**, **for**, **while**, etc.

En relisant le chapitre traitant les variables, nous voyons que **les variables déclarées dans un bloc d'instructions ne sont visibles que dans ce bloc d'instructions.**

La 'vie' d'une variable commence donc à l'endroit de sa déclaration et s'arrête à la fin du bloc dans lequel elle a été déclarée.

Derrière son bloc de déclaration, la variable n'existe plus et on provoque une erreur du type *'cannot find symbol'* si on essaie d'y accéder.

Exemple :

```
public void setLimits(int pMin, int pMax)
{
    if (pMin>pMax)
    {                              ←   ici, la variable help n'existe pas encore
        int help = pMin;           }
        pMin = pMax;               }   dans ce domaine, la variable help est
        pMax = help;               }   accessible.
    }                              ←   ici, la variable help n'existe plus
    min     = pMin;
    max     = pMax;
}
```

Durée de vie de la variable de comptage for :

La variable de comptage est définie à l'intérieur du bloc **for**, donc sa 'vie' commence au début du bloc **for** et s'arrête à la fin du bloc **for**. Derrière la boucle, **i** n'existe plus et on provoque une erreur du type *'cannot find symbol'* si on essaie d'y accéder.

```
...                                ←   ici, la variable i n'existe pas encore
for (int i=0 ; i<=9 ; i++)         }
{                                  }   dans ce domaine, la variable i est
    System.out.println(i);         }   accessible.
}                                  ←   ici, la variable i n'existe plus
...
```

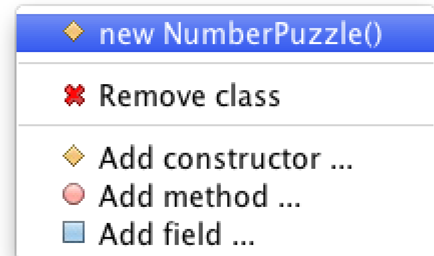
4. Automatisation des tests des classes

Les notions ci-dessous sont intéressantes pour créer des interfaces individuels qui nous permettent d'automatiser les tests des classes développées dans le cours.

4.1. *Création d'objets avec «new»*

Jusqu'ici, vous avez créé des objets en :

1. compilant votre code,
2. en faisant un clic droit sur la classe correspondante et
3. en choisissant « New … » dans le menu contextuel:



La création de nouveaux objets peut aussi se faire dans le code à l'aide du mot clé **new**. En général la syntaxe pour déclarer et initialiser un objet de la classe **XYZ** avec son constructeur par défaut est la suivante :

```
XYZ xyz = new XYZ();
```

Remarque :

En principe, les instances créées ainsi ne seront pas visibles dans l'interface Unimozer. Elles existent dans la mémoire, mais nous ne pouvons pas les "voir" directement.

4.2. *Automatiser les tests de fonctionnement*

Pour automatiser le test du fonctionnement de la classe **Point** décrite dans le chapitre 2 du cours, nous pouvons ajouter une classe 'Test' dans le même projet. Dans la logique du MVC (*Model-View-Controller* → voir cours de 2eGIG), ces classes portent de préférence le suffixe **Controller**. Nous pouvons p.ex. écrire une classe **PointController** comme suit :

```
public class PointController
{
    public void run()
    {
        Point myPoint = new Point(100.5, 150.7);    // création d'une instance
                                                    // de la classe à tester
        System.out.println( myPoint.toString() );
        myPoint.swapCoordinates();                  // tester les méthodes...
        System.out.println( myPoint.toString() );
        myPoint.move(-20.5, 35.3);
        System.out.println( myPoint.toString() );
    }
}
```

Après avoir créé une instance de la classe **PointController**, il suffit d'appeler la méthode **run** pour lancer le test automatisé.

4.3. Démarrage automatique du programme

Pour tester la classe **Point** dans l'exemple précédent, il faut toujours créer une instance de **PointController**, puis appeler manuellement la méthode **run()**. Il serait bien plus pratique si ces deux actions s'effectueraient automatiquement.

En effet, on peut créer en Java une méthode avec la déclaration suivante :

```
public static void main(String[] args)
```

L'effet en sera le suivant :

- On n'aura plus besoin de créer une instance de **PointController** (à cause de **static** dont l'explication apparaîtra bien plus tard dans le cours de programmation...).
- La méthode **main** sera appelée automatiquement lorsqu'on démarre l'application. Oui, en effet, maintenant en ajoutant la méthode **main**, vous avec créé une application que l'on peut démarrer (voir exemple ci-dessous).

En Unimozer, vous pouvez créer une méthode **main** avec une nouvelle classe simplement en cochant la case ☑ Main Method.

Exemple :

Remplacez dans l'exemple précédent la déclaration

```
public void run()
```

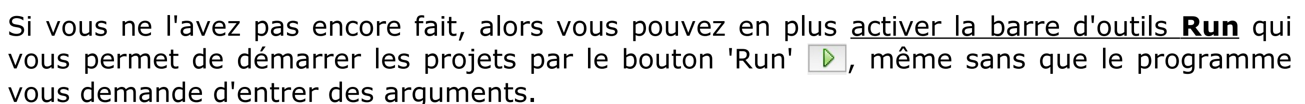
par

```
public static void main(String[] args)
```

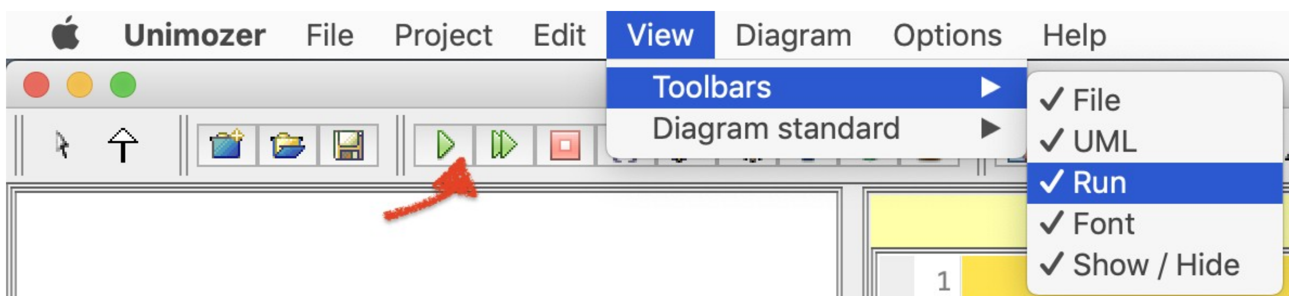
Sauvegardez, puis pour démarrez l'application :

- **En Unimozer :** choisissez **Project → Run** ou poussez la touche **F6**

Dans les deux cas, la méthode **main** s'exécutera et on vous demandera des arguments (→ **args**) pour l'application. Comme nos applications n'ont pas besoin d'arguments de l'extérieur, vous pouvez confirmer simplement en laissant la liste des arguments vide.



Si vous ne l'avez pas encore fait, alors vous pouvez en plus activer la barre d'outils **Run** qui vous permet de démarrer les projets par le bouton 'Run' ▷, même sans que le programme vous demande d'entrer des arguments.



4.4. Saisie de données au clavier en mode texte

Dans le cours, nous avons vu qu'il est possible d'afficher du texte dans la fenêtre des messages, mais il est aussi possible de saisir des données au clavier pendant le déroulement du programme. De cette façon, nous pouvons créer un petit interface texte pour tester nos classes.

Exemple (Ajoutez la classe **PointController** dans le projet de la classe **Point**) :

```
import java.util.Scanner;
public class PointController
{
	public static void main(String[] args)
	{
		Scanner input = new Scanner(System.in);
		System.out.print("Entrez X : ");
		double x = input.nextDouble();
		System.out.print("Entrez Y : ");
		double y = input.nextDouble();

		Point myPoint = new Point(x,y);		// création d'une instance
							// de la classe à tester
		System.out.println( myPoint.toString() );
	}
}
```

Explications

- L'instruction **import java.util.Scanner;** devant la définition de la classe **PointController**, fait en sorte que la classe **Scanner** du paquet **java.util** soit incluse dans ce programme.
- L'instruction **Scanner input = new Scanner(System.in)** crée un objet de la classe **Scanner**[5] qui permet de lire une entrée de l'utilisateur via le clavier.
- La partie **input.nextDouble()** lit ce que l'utilisateur a entré au clavier, l'interprète comme une donnée du type **double** et affecte la valeur lue à la variable **x** respectivement **y**.
- A part **nextDouble**, la classe **Scanner** propose des méthodes pour les autres types numériques vus dans le cours : **nextFloat**, **nextInt**, **nextLong**, **nextByte**, **nextShort**.

5 https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

4.5. Pour avancés : Unimozer.monitor(…,…)

La méthode **`Unimozer.monitor()`** nous permet tout de même d'afficher les instances créées avec **new** dans le panneau des objets, de la même façon que les objets créés "avec la souris".

Utilisation: Indiquer la variable de l'objet et son nom comme arguments après l'avoir créée.

P.ex. **`Unimozer.monitor(xyz, "xyz");`**

Exemple 1 :

```
public class PointController
{
	public static void main(String[] args)
	{
		Point myPoint = new Point(100,150);
		Unimozer.monitor(myPoint, "myPoint"); // afficher l'objet myPoint
		System.out.println( myPoint.toString() );
		myPoint.swapCoordinates();
		System.out.println( myPoint.toString() );
		myPoint.move(-20,35);
		System.out.println( myPoint.toString() );
	}
}
```

myPoint : Point
x = 130.0
y = 135.0

```
Point ( 100.0 , 150.0 )
Point ( 150.0 , 100.0 )
Point ( 130.0 , 135.0 )
```

Exemple 2 :

```
public class PointController
{
	public static void main(String[] args)
	{
		Point point1 = new Point(100,150);
		Unimozer.monitor(point1, "point1"); // afficher l'objet point1
		Point point2 = new Point(200,350);
		Unimozer.monitor(point2, "point2"); // afficher l'objet point2
		// afficher la distance entre les deux points
		System.out.println( point1.calculateDistanceTo(point2) );
	}
}
```

point1 : Point
x = 100.0
y = 150.0

point2 : Point
x = 200.0
y = 350.0

```
223.60679774997897
```